Report Nr. 92.106

Transputer und ihre Architektur
– Eine Übersicht –

von

Martin Malich, Köln

Januar 1992

Mathematisches Institut
Universität zu Köln
Weyertal 86–90
D-5000 Köln 41
Telefon (0221)470–2210
Telex 8882291 unik d
Electronic Mail MM@MI.UNI-KOELN.DE

Transputer und ihre Architektur
– Eine Übersicht –

Martin Malich
Universität zu Köln

26. Januar 1992

Zusammenfassung

In diesem Aufsatz werden die grundlegenden Eigenschaften, sowie die Vor- und Nachteile des Transputerparallelrechners erläutert und Hinweise zur Programmierung und Benutzung des Helios-Betriebssystems gegeben.

Nach einer kurzen Beschreibung der Hardware erläutern wir die Netzwerktopologie eines Transputerrechners. Eine kurze Beschreibung des Helios-Betriebssystems wird im dritten Abschnitt gegeben. Anschließend erfolgt der Einstieg in den Bereich der Kommunikation zwischen Transputerprozessoren innerhalb eines Netzwerks. Im letzten Teil wird speziell auf die Benutzung von *Prozessen* und *Semaphoren* eingegangen, die von der Hardware des Prozessors unterstützt werden.

Die Forderung nach immer mehr Rechenleistung schuf den Ansatz des Parallelrechners. Statt den einzelnen Prozessor durch neue Architekturen und höhere Taktfrequenzen zu beschleunigen, wurde vorgeschlagen möglichst viele Prozessoren zu vernetzen, um so ein Vielfaches an Rechenleistung zu erhalten.
Mittlerweile existieren eine Vielzahl von verschiedenen Parallelrechnerarchitekturen, die sich alle in der Art der Vernetzung, den Prozessoren und dem Speicherzugriff unterscheiden. Einen dieser parallelen Ansätze stellt der des *Transputerparallelrechners* dar. Ein wesentlicher Unterschied zu vielen anderen Parallelrechnerkonzepten ist der lokale Speicher eines Prozessors im Transputerrechner. Dies hat zur Folge, daß die Prozessoren nicht auf einen gemeinsamen Speicher zugreifen können, sondern Datenaustausch nur über spezielle Datenleitungen, den sogenannten Links, möglich ist.

Die effektive Implementierung eines parallelen Ansatzes auf einem Transputerrechner setzt eine intensive Auseinandersetzung mit der zugrunde liegenden Hardware

voraus. Besonders in den Bereichen Kommunikation und Prozeßverwaltung bedarf es einer genaueren Betrachtung, da die Effektivität einer parallelen Anwendung auf einem Transputerrechner sehr stark von der Nutzung dieser Bereiche abhängt.

Schon bei der Entwicklung eines parallelen Programms müssen die Merkmale des Transputers Berücksichtigung finden, sonst kann im schlimmsten Fall das parallele Programm mehr Rechenzeit benötigen, als die sequentielle Version. Dies kann vor allem durch den zusätzlichen Kommunikationsaufwand verursacht werden.

1 Die Transputerkarte (Transputermodul)

Der erste 32-Bit Transputerprozessor T414 wurde 1985 von dem englischen Halbleiterhersteller Inmos auf den Markt gebracht. Es handelt sich dabei um einen VLSI-Chip, den Inmos speziell für den Einsatz in Parallelrechnern entwickelt hat. Dazu wurden die für den Parallelbetrieb wesentlichen Systemroutinen bereits auf der Chip-Ebene implementiert, um eine möglichst hohe Rechenleistung zu erlangen.

Zwei entscheidende Leistungsmerkmale im Unterschied zu herkömmlichen Mikroprozessoren bestehen in der Kommunikationsfähigkeit über sogenannte Links und dem virtuellen Multiprocessing (process scheduling), das im Chip implementiert wurde.

Der Transputer T800 gilt als der größere Bruder des T414. Er besitzt einen größeren OnChip-Memory und eine eingebaute Fließkomma-Arithmetik.

Daten zum Transputer T800:

- 32 Bit RISC-Prozessor mit 20 MHz Taktfrequenz und 4 KB OnChip-Memory.
- 12 MIPS, 1.9 MFLOPS Rechenleistung mit eingebauter Floatingpoint-Unit.
- 20 MBit/s Datenübertragungsrate (bidirektional) über Links.

Ein Transputerrechner besteht aus einzelnen Modulen, den *Transputerkarten* bzw. *Transputermodulen*. Abbildung 1 zeigt den Aufbau einer solchen Karte. Dabei erkennt man, daß zum T800 nur sehr wenig zusätzliche Hardware benötigt wird, um ein komplettes Transputermodul zu erhalten.

Der RISC-Prozessor besitzt einen sehr kompakten und optimierten Befehlssatz, denn es wurde versucht die am häufigsten in Programmen benutzten Befehle mit möglichst wenig Code und kurzer Ausführungszeit zu implementieren. Tatsächlich gelang es Inmos *die* Assembler-Befehle, die etwa 70 Prozent eines Transputerprogrammes ausmachen, in einem Byte Code unterzubringen. Viele dieser Befehle benötigen zur Ausführung nur einen Taktzyklus.

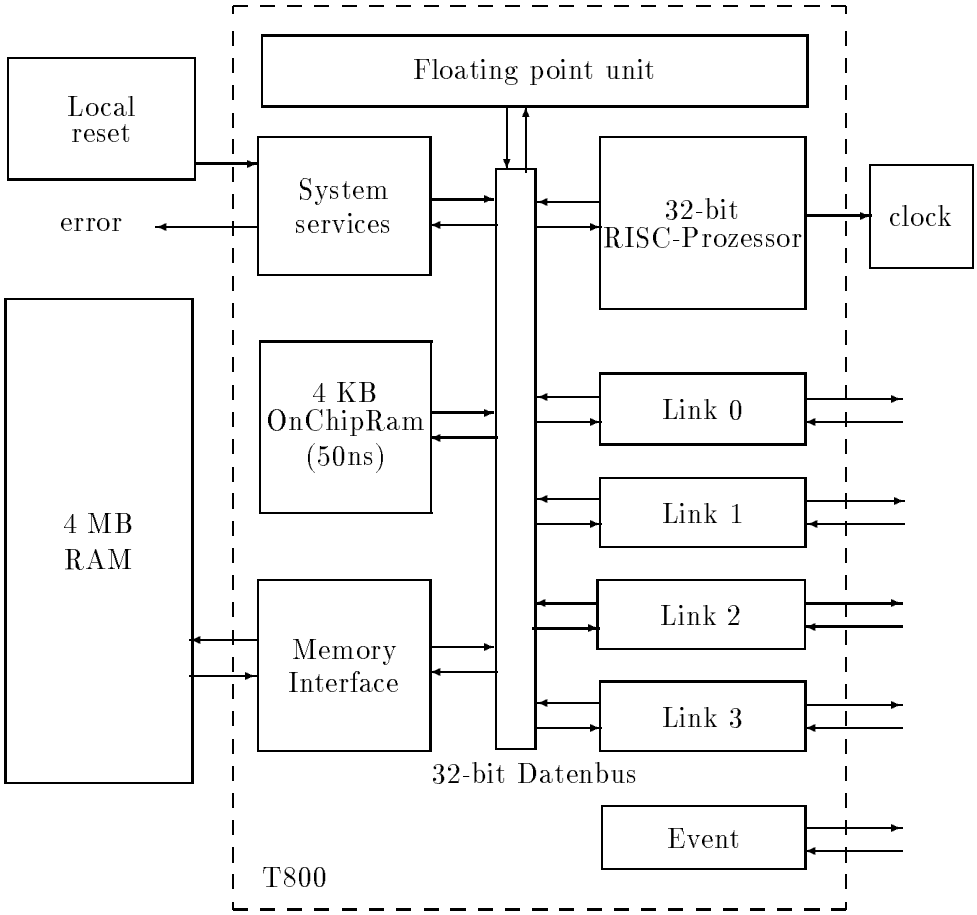

Abbildung 1: Der Aufbau einer Transputerkarte

Der *OnChip-Memory* kann von eigenen Programmen benutzt werden, um den Zugriff des Prozessors auf den großen Hauptspeicher zu vermeiden. Weil der Transputer nur 3 Rechenregister besitzt, müssen häufig Variablen aus dem Speicher geladen werden. Die Zugriffszeit des Prozessors auf den OnChip-Memory entspricht nur einem Taktzyklus, also bei einer Taktfrequenz von 20 MHz genau 50 ns. Je nach Art und Größe des externen Speichers benötigt der Zugriff dort die etwa 3 bis 5-fache Zeit. Daher ist es sehr günstig den Variablenbereich im Prozessor-Ram anzulegen. Falls die auszuführende Routine klein genug ist, kann sie komplett ins OnChip-Ram kopiert werden, um die Ausführungszeit zu verringern.

Die Speichereinheit der Transputerkarte kann nur von dem lokalen Transputerprozessor angesprochen werden. Ein Transputerrechner besitzt also im Gegensatz zu anderen Parallelrechnerkonzepten kein *shared memory*. Die typische Speichergröße beträgt 4 MB.[1]

Weil der Datenaustausch zwischen den Prozessoren über einen gemeinsamen Speicher nicht möglich ist, wurde die Fähigkeit der *point-to-point* Datenübertragung implementiert. Jede Karte verfügt dazu über 4 Links, die jeweils zum Lesen und Schreiben von Daten benutzt werden können. Dabei erfolgt der Datentransfer durch einen direkten Zugriff auf die Speichereinheit (DMA: direct memory access). Dies bedeutet, daß die Daten beim Lesevorgang ohne Zwischenspeicherung an die vorgegebene Zieladresse ins RAM geschrieben werden und beim Schreibvorgang direkt aus dem Speicher ausgegeben werden. Das Verschicken der Daten geschieht byteweise und synchronisiert.

Die maximale bidirektionale Datenübertragungsrate beträgt beim T800 theoretisch 20 MBit/s. Bei unidirektionaler Übertragung erreicht man unter Benutzung des Helios-Betriebssystems maximal etwa 9 MBit/s. Dieser Wert ist aber sehr stark abhängig von der Größe des verschickten Datenpakets.[2]

Bei den bisherigen Parallelrechnerkonzepten, die auf einem gemeinsamen Speicher basieren, können Engpässe auf dem Datenbus entstehen, falls die Anzahl der Prozessoren und damit der Zugriff auf den Speicher zunimmt. Dagegen kann aufgrund der point-to-point Kommunikation die Anzahl der Prozessoren bei Transputerrechnern beliebig groß sein, ohne daß derartige Engpässe entstehen. Natürlich muß man auch bedenken, daß bei einem Transputerrechner die Entfernung zwischen zwei Prozessoren, die Daten austauschen wollen, größer werden kann, wenn die Anzahl der Transputer zunimmt.

[1] Es sind auch Module mit 1, 2 oder 8 MB erhältlich.

[2] Siehe dazu auch Tabelle 1 auf Seite 11.

2 Das Transputernetzwerk

Der Aufbau eines Transputerparallelrechners vollzieht sich im wesentlichen durch das paarweise Verbinden von Links verschiedener Transputerkarten. Auf diese Art und Weise entsteht ein Netzwerk, dessen Knoten den Transputerkarten entsprechen und dessen Kanten die Verbindung von je zwei Links darstellen.

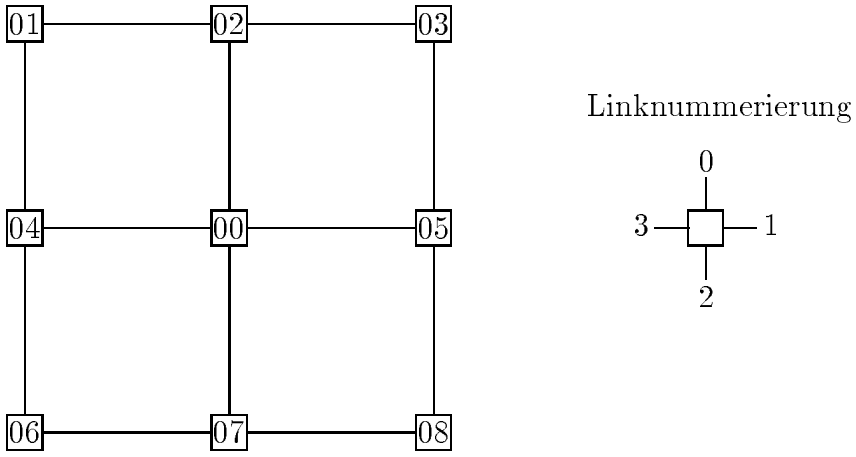

Abbildung 2: Ein Transputernetzwerk mit 9 Prozessorknoten

Abbildung 2 zeigt ein Netzwerk mit 9 Knoten in Form eines Gitters. Die Transputerkarten sind dabei fortlaufend durchnummeriert. So ist zum Beispiel Prozessor 00 über den Link 3 mit dem Link 1 des Prozessors 04 verbunden.

Die auszuwählende Topologie stellt immer einen ungerichteten Graphen dar, der einen maximalen Knotengrad von höchstens 4 besitzt. Mehrfachkanten und Schleifen sind zwar zugelassen, aber selten sinnvoll. Durch die gegebene Anzahl zur Verfügung stehender Links ist es nur für höchstens 5 Prozessoren möglich einen vollständigen Graphen als Transputernetzwerk zu konfigurieren.

Ein Schritt bei der Erstellung paralleler Anwendungen auf einem Transputerrechner ist immer die Auswahl eines geeigneten Netzwerkes. Da 4 Links pro Prozessor eine starke Restriktion bedeuten, ist es bei vielen Anwendungen nicht möglich je zwei Prozessoren, die Daten austauschen sollen, über einen Link miteinander zu verbinden.

Das Einrichten der gewünschten Topologie stellt offensichtlich ein Problem dar, da es höchst unkomfortabel wäre, wenn man die Linkverbindungen beim Wechseln des Netzwerkes umstecken müßte. Zur Lösung werden von verschiedenen Herstellern[3] Transputerumgebungen angeboten, in die eine begrenzte Anzahl von

[3]In Deutschland vor allem Parsytec, in England Meico, in Frankreich Telmat.

Transputermodulen integriert werden kann. Auch ist es möglich mit mehreren Benutzern gleichzeitig an einem Transputerrechner zu arbeiten. Dabei können die zur Verfügung stehenden Prozessoren unter den Benutzern aufgeteilt werden. Ein Transputer kann zwar auch von mehreren Anwendern gleichzeitig benutzt werden, doch können dadurch gewisse Gefahren und Nachteile entstehen.

Der diesem Aufsatz zugrunde liegende Transputerrechner der Firma Parsytec besteht im wesentlichen aus 32 Transputerkarten, die mit dem T800 Prozessor bestückt sind und über 4 MB Speicherplatz verfügen. Zusätzlich existieren noch Module für besondere Aufgaben. [4] Maximal 8 Benutzer können direkt an die Transputerumgebung angeschlossen werden.

Ein wichtiger Bestandteil dieser Umgebung ist die *Network-Configuration-Unit (NCU)*. Dabei handelt es sich um einen kleinen Transputerrechner, der mit einem Multiplexer verbunden ist, welcher für das Zusammenschalten der Links zuständig ist.
Fordert der Benutzer eine bestimmte Netzwerktopologie, so schickt er diese als Binärcode an den *Network-Configuration-Manager (NCM)*. Dieses Programm befindet sich auf der NCU und sorgt dafür, daß der Multiplexer die in der geforderten Topologie angegebenen Linkverbindungen herstellt. Durch die große Anzahl von insgesamt 128 Links bei 32 Prozessoren ist es nur möglich Links mit gleicher Parität zu verbinden.

3 Das Betriebssystem Helios

Im Jahr 1989 erschien das erste Multiprozessor Transputer-Betriebssystem Helios. Das auf Forschungsergebnissen der Universität von Cambridge und anderen Hochschulen basierende Konzept wurde von der englischen Firma Perihelion in Helios umgesetzt. Es ist weitgehend Unix kompatibel und bietet dem Programmierer eine geeignete Entwicklungsoberfläche.

Außer dem Betriebssystem Helios, existieren zur Erstellung von Programmen auf einem Transputerrechner eine Reihe von Entwicklungsumgebungen. Dazu gehören z.B. das *Transputer Development System (TDS)*, das von Inmos entwickelt wurde, *CS-Tools* von Meiko Ltd. und die von Parsytec modifizierte Version des TDS namens *Multitool*.

Im Unterschied zu diesen Entwicklungswerkzeugen benötigt man unter Helios keine spezielle für die parallele Programmierung entwickelte Sprache wie Occam oder Parallel C, sondern kann mit einem gewöhnlichen Compiler für Fortran, Pascal, Modula oder Ansi-C arbeiten.

[4] Fileserver, Graphicserver und Ethernet.

Während die Entwicklungsumgebungen mit 'nackten' Transputerprozessoren arbeiten, stellt Helios auf jedem Prozessor einen sogenannten *Nucleus* zur Verfügung, der wichtige Routinen zur Kommunikation enthält. Zusammen mit dem Nucleus erleichtert Helios dem Programmierer die Arbeit, weil das Routing im Transputernetzwerk komplett vom Betriebssystem gesteuert wird.

Ein zu erwähnender Nachteil von Helios ist der, durch den Verwaltungsaufwand entstehende *overhead*, der sich vor allem bei der Kommunikation mittels kleiner Datenpakete bemerkbar macht. Dieser Nachteil kann nur durch Benutzung der Helios *low-level Routinen* umgangen werden.

4 Die Kommunikation

Im Gegensatz zur sequentiellen Programmierung entstehen bei parallelen Anwendungen auf einem Transputerrechner üblicherweise Kommunikationskosten, die stets einen zusätzlichen Zeitaufwand bedeuten. Um die Geschwindigkeitsvorteile eines Mehrprozessorsystems nicht durch die anfallenden Kommunikationskosten zu verlieren, müssen die Kommunikationswege kurz und die zu verschickende Datenmenge möglichst klein gehalten werden. In diesem Zusammenhang gilt es immer zu prüfen, ob der entstehende Kommunikationsaufwand nicht größer ist, als der durch die Parallelisierung erzielte Rechenzeitgewinn.

4.1 Die parallele Minimumbestimmung

Das nun folgende Beispiel eines parallelen Algorithmus soll zeigen, daß nicht jeder in der Theorie sehr erfolgversprechende Algorithmus auch effizient auf einem Transputerrechner implementiert werden kann.

Zur parallelen Minimumbestimmung von n natürlichen Zahlen läßt sich die Methode des *balanced binary tree* nutzen:

Seien $a_1, \ldots, a_n \in I\!N$ mit $n = 2^k \in I\!N, k \in I\!N$ gegeben.
Gesucht ist das Minimum $m = \min_{1 \leq i \leq n} a_i$.

Sei nun ein Transputernetzwerk mit $(2n - 1)$ Prozessoren gegeben, die in einem binären Baum angeordnet sind. Die n Blätter des Baumes stellen die Starttransputer dar, denen jeweils eine Zahl a_i zugeordnet wird.
Jeder Prozessor liest nun von seinen beiden Söhnen im Baum zwei Zahlen a und b, führt einen Vergleich durch und schickt die kleinere der beiden Zahlen an seinen Vater. Dann besitzt die Wurzel nach k Iterationen das Minimum m. Die Abbildung 3 zeigt den Ablauf des Verfahrens für 8 Zahlen.

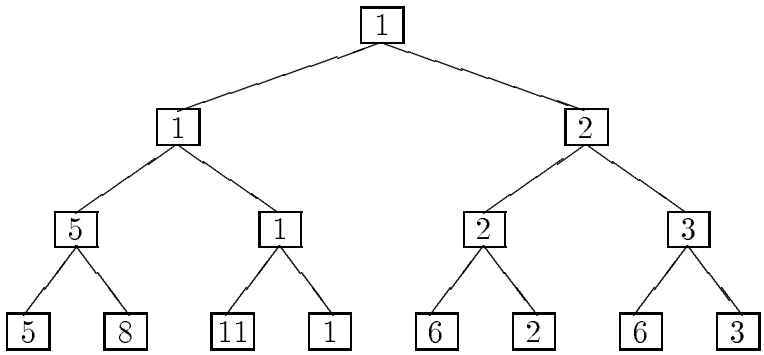

Abbildung 3: Die parallele Minimumbestimmung von 8 Zahlen

Anzahl der parallelen Vergleichsschritte (Tiefe des Baumes):	$\log n$.
Anzahl der parallel verschickten Zahlen:	$2 \log n$.

Sei nun t_v die benötigte Zeit für einen Vergleich und t_k die Zeit für das Verschicken einer Zahl a_i. Dann läßt sich die für den sequentiellen Algorithmus benötigte Zeit T_{seq} und die Zeit T_{par}, die für die parallele Version benötigt wird, wie folgt berechnen:

$$T_{seq} = t_v(n-1), \qquad T_{par} = t_v \log n + 2t_k \log n$$

Man kann sich nun Fragen in welchem Verhältnis t_v und t_k zueinander stehen müssen, damit durch die Parallelisierung ein Speedup s entsteht:

$$\begin{aligned} T_{par} &\leq \frac{1}{s} T_{seq} \\ \Rightarrow t_v \log n + 2t_k \log n &\leq \frac{n-1}{s} t_v \\ \Rightarrow t_k &\leq \frac{1}{2} t_v \left(\frac{n-1}{s \log n} - 1 \right) \end{aligned} \tag{1}$$

Geht man davon aus, daß ein Vergleich etwa 10 Prozessor-Taktzyklen benötigt, so ergibt sich bei einer Taktfrequenz von 20 MHz eine Zeit von $t_v = 5 * 10^{-7}$ Sekunden. Wenn die zu verschickende Zahl eine Größe von 4 Bytes besitzt, benötigt Helios dafür eine Kommunikationszeit von $t_k = 10^{-3}$ Sekunden. Fordert man einen Speedup von 1, d.h. die parallele Version benötigt dieselbe Zeit, wie das sequentielle Programm, so folgt aus der obigen Beziehung (1): $n > 63000$. Dafür müßte der Transputerrechner aber über 120000 Prozessoren zur Verfügung stellen, was derzeit technisch und finanziell kaum möglich ist.

Dieses triviale Beispiel zeigt, daß eine bzgl. der Rechenkomplexität erfolgversprechende Parallelisierung noch kein Garant für eine tatsächliche Rechenzeitverkürzung ist. Denn es muß zusätzlich darauf geachtet werden, daß die Prozessoren zwischen den Kommunikationen mit anderen Prozessoren genügend Arbeit zum Rechnen besitzen. Eine Parallelisierung wird nur dann erfolgreich sein, wenn die Kommunikation im Verhältnis zur Menge der lokal durchgeführten Berechnungen gering ist.

Viele in der Theorie sehr effiziente Parallelisierungen bekannter Algorithmen würden bei der Implementierung auf einem Transputerrechner sehr schlecht abschneiden. Die Gründe dafür sind dieselben wie bei der parallelen Minimumbestimmung:

- Bei kleinen Problemen stellt die Kommunikation einen zu großen overhead dar.
- Bei großen Problemen stehen die benötigten Prozessoren nicht zur Verfügung.

Durch die neue Transputergeneration in Form des T9000 werden die bestehenden Probleme noch vergrößert, da sich die Rechenleistung im Vergleich zum T800 etwa verzehnfacht hat, während die Datenübertragung nur um den Faktor 4 schneller geworden ist.[5]

4.2 Routing im Prozessornetzwerk

In einem Netzwerk mit mehr als 5 Knoten ist es häufig nötig Daten nicht nur an einen der benachbarten Prozessoren, sondern auch an weiter entfernte Transputer zu verschicken. Dazu benötigt man auf jedem Prozessor im Netzwerk ein Programm, das Daten empfängt und sie in Richtung Zielprozessor weiterleitet. Diesen Vorgang nennt man auch *Routing* und das zugrunde liegende Routingsystem den *Router*.

Das Transputer-Betriebssystem Helios stellt einen Router zur Verfügung, der eine zu verschickende Nachricht selbständig auf dem kürzesten Weg durch das Netzwerk ans Ziel schickt.

Will ein Prozessor eine Nachricht verschicken, so wird zunächst in einer Namentabelle auf diesem Prozessor nachgeschaut, ob für den Zielprozessor bereits ein Eintrag existiert. Falls es einen Eintrag gibt, so spezifiziert dieser den Weg zum Zielobjekt und die Nachricht kann über den entsprechenden Link weitergeleitet

[5] Der T9000 stellt den Nachfolger des T800 Transputers dar. Er soll Ende 1992 auf dem Markt erhältlich sein.

Paketgröße in KB	1-Link in KB/Sek	2-Link in KB/Sek	3-Link in KB/Sek
0.05	25	16	12
0.2	96	62	46
0.5	365	142	114
2	560	390	334
5	825	588	535
10	981	712	669
20	1082	800	767
40	1141	850	827
80	1172	876	857
160	1172	879	864
320	1172	879	866
640	1176	883	870

Tabelle 1: Kommunikationsraten beim Datentransfer unter Helios

werden. Ist kein Eintrag vorhanden, so werden vom Quellprozessor ausgehend, Nachrichten an alle benachbarten Prozessoren verschickt, die wiederum Anfragen an alle angrenzenden Knoten stellen. Die so entstehende *breadth-first-search* wird abgebrochen, sobald der Zielprozessor erreicht wurde oder ein Prozessor einen passenden Eintrag in seiner Namentabelle besitzt. Alle auf dem so gefundenen Weg liegenden Transputer können nun den Pfad in ihre Namentabelle eintragen und ihn für spätere Anfragen bereithalten.

4.3 Die Datenübertragungsrate

Muß ein Datenpaket, um an seinen Bestimmungsort zu gelangen, über mehr als einen Link geschickt werden, so verringert sich die Datenübertragungsrate beträchtlich (siehe dazu Tabelle 1). Desweiteren erkennt man, daß es wenig vorteilhaft ist kleine Datenpakete zu verschicken, da der anteilmäßig sehr hohe Verwaltungsaufwand die Datenübertragungsrate stark vermindert. Bei Anwendungen, die auf das hoch frequentierte Verschicken von kleinen Datenmengen nicht verzichten können, besteht noch die Möglichkeit des *dumben* von Links. Dabei werden die Links unter Umgehung des Helios Betriebssystems in den *dumb* Modus geschaltet und die Daten direkt über den Link zum nächsten Prozessor geschickt. Man erhält dadurch bei kleinen Datenpaketen einen Geschwindigkeitsvorteil, muß sich aber beim Verschicken an weiter entfernte Prozessoren ein eigenes Routingsystem aufbauen. Darüberhinaus birgt das Umgehen von Helios und dessen Unix-kompatible

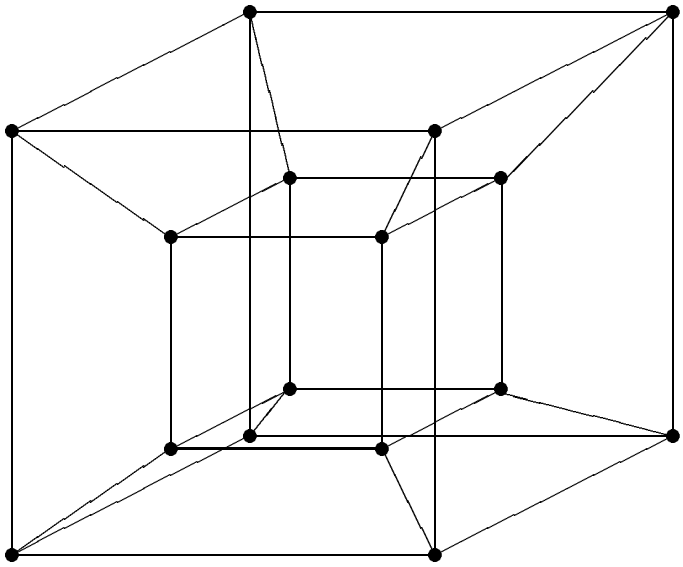

Abbildung 4: Ein 4-dimensionaler Hyperwürfel

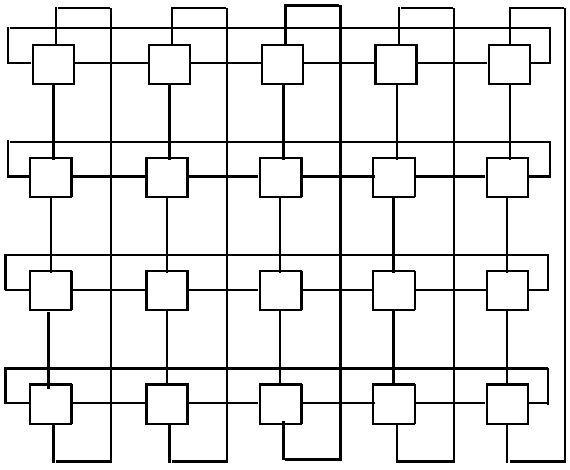

Abbildung 5: Ein 4×5-Toroid

POSIX-Library[6] gewisse Gefahren wie 'böse' Programmabstürze und Kompatibilitätseinbußen.

Eine Möglichkeit die Kommunikationswege möglichst kurz zu halten ist die Auswahl einer geeigneten Netzwerktopologie. Die Eignung einer Topologie hängt sehr stark von dem gegebenem Problem ab. Allgemein sind aber Graphen mit kleinem Diameter zu bevorzugen, wie z.B. ein $n \times m$-Toroid mit nm Knoten oder ein Hyperwürfel der Dimension 3 oder 4 mit 8 bzw. 16 Knoten.

Abbildung 4 zeigt einen 4-dimensionalen Hyperwürfel mit 16 Knoten und Diameter 4. In Abbildung 5 ist ein 4×5-Toroid mit 20 Knoten und ebenfalls Diameter 4 dargestellt. Der $n \times m$-Toroid besitzt nicht nur den Vorteil eines kleinen Diameters, sondern ist auch sehr variabel was die Anzahl der enthaltenen Prozessoren betrifft. Allgemein kann der Durchmesser eines Toroids wie folgt berechnet werden:

Satz (Toroid-Diameter)

Beh.: Der Diameter des $n \times m$ Toroid beträgt

$$d = \left\lfloor \frac{n}{2} \right\rfloor + \left\lfloor \frac{m}{2} \right\rfloor .$$

Bew.: Die Prozessoren seien in Form eines Gitters angeordnet und in der für Matrizen üblichen Weise nummeriert. Dabei sei n die Anzahl der Zeilen und m die Anzahl der Spalten.

Es seien zwei Prozessoren $P_1 = (i, j)$ und $P_2 = (k, l)$ durch ihre Position im Gitter gegeben. Zu zeigen ist, daß ein Weg von P_1 nach P_2 existiert, der höchstens d viele Kanten benutzt.

(1) Es existiert ein Weg von $P_1 = (i, j)$ nach $P_1^* = (k, j)$, der höchstens $\lfloor n/2 \rfloor$ Kanten benutzt.
Denn: Die Prozessoren $(1, j), \ldots, (n, j)$ stellen zusammen mit den zugehörigen Kanten einen Ring bestehend aus n Prozessoren dar. Da ein Ring mit n Prozessoren offensichtlich den Diameter $\lfloor n/2 \rfloor$ besitzt, ist (1) gezeigt.

(2) Es existiert ein Weg von $P_1^* = (k, j)$ nach $P_2 = (k, l)$, der höchstens $\lfloor m/2 \rfloor$ Kanten benutzt.
Denn: Analog zu (1).

Aus (1) und (2) erhält man einen Weg mit der gewünschten maximalen Länge d.

Es ist nun noch zu zeigen, daß in jedem $n \times m$-Toroid zwei Knoten existieren, deren kürzester Weg aus mindestens d Kanten besteht. Wähle dazu die Knoten

$$P_1 = (1, 1) \text{ und } P_2 = (\lfloor n/2 \rfloor + 1, \lfloor m/2 \rfloor + 1).$$

[6] Die POSIX-Library enthält wichtige Routinen zur Ein/Ausgabe über *Streams*.

In jedem Weg zwischen P_1 und P_2 müssen mindestens $\lfloor n/2 \rfloor$ waagerechte und $\lfloor m/2 \rfloor$ senkrechte Kanten benutzt werden. Dabei heißt eine Kante der Form $((i,j),(i,k))$ waagerecht und entsprechend $((i,j),(k,j))$ senkrecht. Also folgt wie gewünscht, daß jeder kürzeste Weg von P_1 nach P_2 mindestens d Kanten enthält.

□

4.4 Das Mapping

Die parallele Anwendung auf einem Transputersystem wird als *Taskforce* bezeichnet. Die einzelnen *Tasks* einer Taskforce sind in sich abgeschlossene Programme, die über sogenannte *Streams* (Datenströme) miteinander kommunizieren können. Die Streams entsprechen aber nicht den Hardwarelinks der Transputerprozessoren, sondern stellen softwaremäßig eingerichtete Datenkanäle dar. So können auch zwei nicht benachbarte Transputer über *einen* gemeinsamen Stream miteinander kommunizieren.

Man unterscheidet daher zwei verschiedene Topologien:

Hardware-Topologie ist die, durch das paarweise Verbinden von Transputer-Links entstehende Topologie.[7]

Software-Topologie besteht aus Tasks und Streams und wird durch die Taskforce bestimmt.[8]

Insbesondere existieren keinerlei Einschränkungen bzgl. der maximalen Valenz eines Taskknotens der Softwaretopologie. Die Anzahl der Knoten innerhalb der Softwaretopologie darf die Anzahl der Knoten in der Hardwaretopologie durchaus überschreiten. In diesem Fall müssen mehrere Tasks auf einem Transputerprozessor ausgeführt werden.

Unter einem Mapping versteht man eine Abbildung der Tasks und Streams einer Taskforce auf die Prozessoren und Links des Transputerrechners. Entspricht die Softwaretopologie der Taskforce genau der Hardwaretopologie des Transputerrechners, so existiert ein Mapping, das eine bijektive Zuordnung zwischen Streams und Links bzw. Tasks und Prozessoren herstellt. Ein solches Mapping wird bijektiv genannt.
Falls es Tasks gibt, die mehr als 4 Streams benutzen oder wenn die Softwaretopologie von der Hardwaretopologie abweicht, so ist u.U. ein bijektives Mapping nicht mehr möglich. Es entsteht nun das Problem die Tasks so auf die Prozessoren zu

[7] Unter Helios wird die Hardware-Topologie über sogenannte *Resource-Maps* definiert.

[8] Die Software-Topologie wird in Helios unter Benutzung der Programmiersprache *CDL* (component distribution language) definiert.

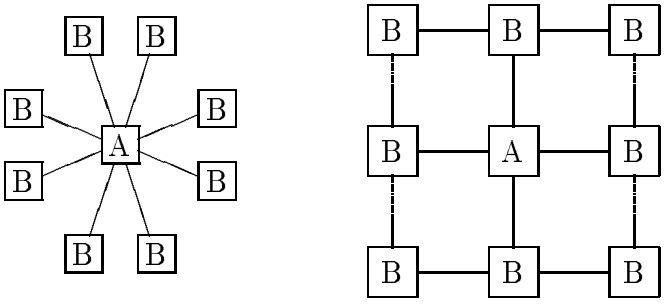

Abbildung 6: Eine Taskforce mit zugehörigem Mapping

verteilen, daß die Streams möglichst wenige Links benutzen müssen, um Datenpakete vom Starttransputer an den Zielort zu schicken. Sind weniger Prozessoren als Tasks vorhanden, so müssen mehrere Tasks zusammen auf einem Prozessor ausgeführt werden, was im Mapping ebenfalls berücksichtigt werden muß. Abbildung 6 zeigt die Software-Topologie einer Taskforce und ein zugehöriges Mapping auf die Hardware-Topologie mit 9 Prozessoren in Form eines Gitters.

5 Die Prozesse

Jede Task stellt einen Prozeß dar, der eine beliebige Anzahl von weiteren Prozessen starten kann, welche alle pseudo-parallel auf einem Transputer ausgeführt werden. Im Unterschied zu Tasks, die auf unterschiedlichen Prozessoren gestartet werden können, sind Prozesse an den Transputer des aufrufenden Prozesses gebunden. Insbesondere folgt daraus, daß jeder dieser Prozesse Zugang auf den gemeinsamen Speicher besitzt.

Im Gegensatz zu einem Multitasking-Betriebssystem wie Unix, wird der *Verteilungsmechanismus (Process Scheduling)* nicht vom Betriebssystem, sondern von der Hardware auf dem Transputerchip durchgeführt. Dies hat zwar den Nachteil, daß es nur zwei verschiedene Prioritäten für Prozesse gibt, dafür beträgt aber die Umschaltzeit zwischen den Prozessen weniger als eine Mikrosekunde.

Jeder Prozeß besitzt einen der folgenden Zustände:

- aktiv
  - wird gerade ausgeführt
  - bereit
- passiv

- wartet auf Eingabe
- wartet auf Ausgabe
- wartet bis zu einem bestimmten Zeitpunkt
- angehalten

Für das *Scheduling* der Prozesse wurde im Transputerchip der *Prozeßverwalter* implementiert. Dieser führt eine Liste (Prozeßliste), in der jeder Prozeß vom Status `bereit` einen Eintrag besitzt. Soll ein neuer Prozeß gestartet werden, wird seine Speicheradresse an das Ende der Prozeßliste gehängt. Der aktuelle Prozeß wird solange ausgeführt, bis er an seinem normalen Ende angelangt ist oder descheduled wird. In beiden Fällen startet der Prozeßverwalter anschließend den nächsten Prozeß aus der Liste und entfernt dort dessen Eintrag.

input message	output message	output byte	output word
timer alt wait	timer input	stop on error	alt wait
jump	loop end	end process	stop process

Tabelle 2: Descheduling Points Instruktionen

Der aktuelle Prozeß kann nur bei bestimmten Befehlen vom Prozeßverwalter unterbrochen werden. Diese Befehle sind bekannt als sogenannte *Descheduling Points*. In Tabelle 2 sind alle ausgezeichneten Instruktionen aufgeführt.

Erst wenn ein Prozeß länger als zwei *Zeitscheibenintervalle (time slices, etwa 2 ms)* ausgeführt wurde, muß er beim nächsten Descheduling Point unterbrochen werden. Weil die einzigen Befehle mit denen sich Programmschleifen konstruieren lassen, die Instruktionen *jump* und *loop end* sind, kann es nicht passieren, daß ein Prozeß beliebig lange ausgeführt wird.

Die Reaktion des Prozeßverwalters auf das Descheduling, hängt von der Instruktion ab, welche die Ausführung des aktuellen Prozesses unterbrochen hat:

- Die aktuell ausgeführte Instruktion gehört zu den Befehlen *jump* oder *loop end*.
  $\Rightarrow$ Der Prozeß bekommt den Status `bereit` und wird an das Ende der Prozeßliste gehängt.
- Die aktuelle Instruktion gehört zu den Befehlen *input message, output message, output byte, output word, alt wait, timer alt wait, timer input*.

⇒ Der Prozeß wird `passiv`. Er wird nicht erneut in die Prozeßliste aufgenommen und unterliegt nicht mehr dem Prozeß Scheduling. Seine Verwaltung wird fortan von anderen Hardwareeinheiten des Transputers übernommen.[9]

- Bei einer der beiden Instruktionen *stop process* oder *stop on error* wird der Prozeß in den `halt`-Zustand versetzt. Er kann nur durch *start process* erneut in die Prozeßliste aufgenommen werden.
- Durch *end process* wird der aktuelle Prozeß ordentlich beendet.

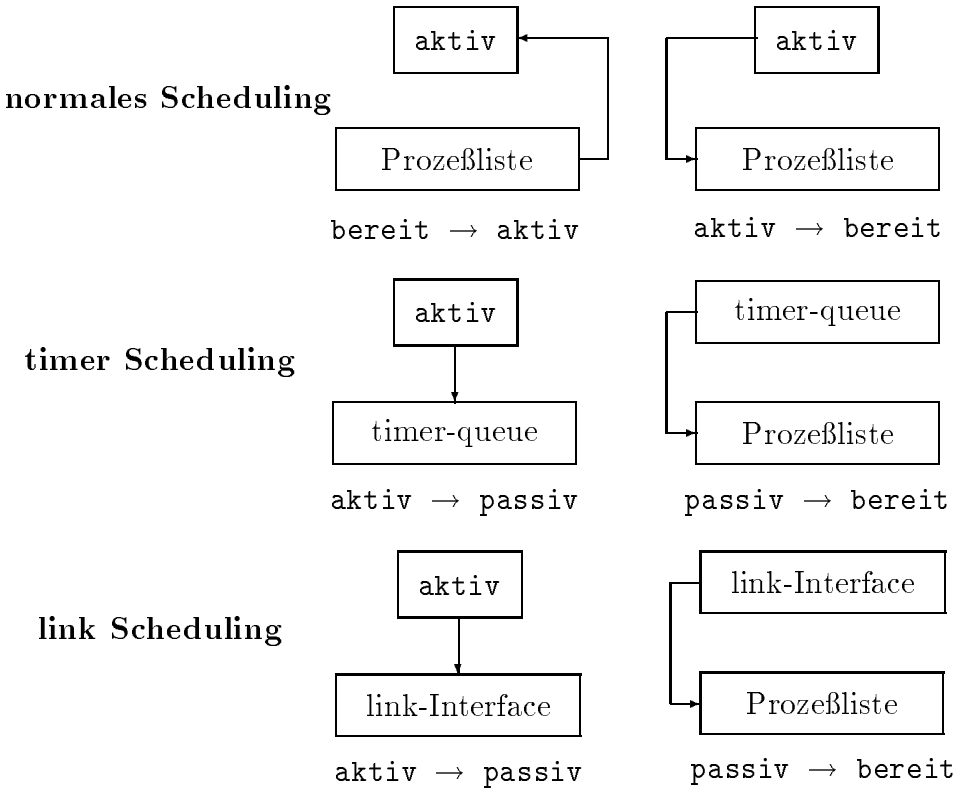

Abbildung 7: Das Scheduling der Prozesse

Abbildung 7 zeigt die Übergänge zwischen verschiedenen Zuständen der Prozesse während des Scheduling. Bei der Passivierung eines Prozesses durch einen der timer-Befehle wird der Prozeß in eine spezielle Zeitgeberschlange (timer queue) eingereiht, in der er für die verlangte Zeit verweilen muß. Überschreitet der Timer

[9] Zu den Hardwareeinheiten gehören der Timer und die Linkeinheit.

die vorgegebene Startzeit des ersten Prozesses in der timer queue, bekommt der Prozeß den Status `bereit` und wird in die Prozeßliste eingefügt.

Die Befehle *input message* und *output message* dienen der Übertragung von Daten über einen der 4 Links. Nach der Passivierung des Prozesses übernimmt das Link-Interface die Kontrolle und wartet bis das Link des anderen Transputers fertig ist. Sobald beide Links bereit sind, findet der eigentliche Datenaustausch statt. Nach Beendigung wird der Prozeß erneut in die Prozeßliste eingefügt.

Weil die `passiven` Prozesse nicht vom Prozeßverwalter, sondern von anderen Hardwareeinheiten gesteuert werden, benötigen sie keinerlei Rechenzeit vom Prozessor. Bemerkenswert ist auch, daß die Links vollkommen unabhängig und parallel zum Prozessor arbeiten. Sie besitzen DMA-Zugriff auf den gesamten Adreßraum des Transputers, dies gilt für den internen als auch für den externen Speicher. Zusammen mit der sehr effektiven Implementierung des Prozeß Scheduling auf Chip-Ebene bedeutet dies für den Programmierer, daß vor allem die Routinen zur Ein/Ausgabe über Links als Prozesse zu implementieren sind, um den übrigen Programmablauf so wenig wie möglich durch die Kommunikation zu belasten.

5.1 Die Implementierung von Schreib- und Leseprozessen

Weil ein `passiver` Prozeß keinerlei Rechenzeit benötigt, können die Schreib- und Lesevorgänge sehr effektiv als Prozesse programmiert werden. Zur Synchronisierung des Ablaufs wurden unter Helios *Semaphoren* implementiert.

5.1.1 Semaphoren

Der Begriff der Semaphore[10] wurde von E.W.Dijkstra eingeführt, um den gemeinsamen Zugriff von Prozessen einer Multitaskingumgebung auf ihre Resourcen (z.B. den Speicher) zu synchronisieren.

Eine Semaphore läßt sich durch zwei Grundoperationen nutzen. Durch den *Wait*-Befehl hält der Prozeß solange an, bis die Semaphore durch den *Signal*-Befehl freigegeben wird. Wichtig ist dabei, daß die Befehle, welche die Semaphore beeinflussen, *unteilbar* sind. Das heißt, daß der Prozeßverwalter während eines solchen Befehls nicht auf einen anderen Prozeß umschalten darf.
Falls mehr als ein Prozeß auf die Freigabe einer Semaphore warten, wird nach der Ausführung des Signal-Befehls nur der Prozeß fortgesetzt, der die längste Zeit wartet.

Neben den meisten anderen Multitasking-Betriebssystemen wurden auch unter Helios Semaphoren implementiert. Jede Semaphore unter Helios besteht aus einem

[10] griechisch: Zeichenträger

Zähler und zwei *Zeigern*. Die beiden Zeiger stellen jeweils den Anfang und das Ende eine Schlange (Queue) dar. Jedes Element in dieser Schlange beinhaltet die Adresse des als nächstes auszuführenden Befehls eines Prozesses.

Es existieren unter Helios vier verschiedene Befehle mit denen sich Semaphoren nutzen lassen:

InitSemaphore(s, w) **:** Setzt den Zähler der Semaphore s auf den Wert w und die beiden Zeiger auf *NIL (NULL)*.

Wait(s) **:** Der Zähler der Semaphore s wird um eins dekrementiert. Falls der Zähler danach kleiner als Null ist, wird die Adresse des nächsten Befehls als neues Element an die Queue der Semaphore s gehängt. Der Zeiger auf das Ende der Semaphore wird entsprechend verändert. Anschließend wird der Prozeß angehalten und vom Prozeßverwalter nicht mehr in die Prozeßliste eingetragen.
Ist der Zähler größer oder gleich Null, so wird der Ablauf des Prozesses normal fortgesetzt.

Signal(s) **:** Der Zähler der Semaphore s wird um eins inkrementiert. Falls sich ein Prozeß in der Queue der Semaphore befindet, wird der erste dieser Prozesse aus der Schlange entfernt und als Prozeß vom Status `bereit` in die Prozeßliste des Prozeßverwalter eingetragen.

w=TestSemaphore(s) **:** Nach Ausführung dieses Befehls enthält die Variable w den Wert des Zählers der Semaphore s.

Anhand der folgenden Schreib/Lese-Prozesse wird ersichtlich, wie man Semaphoren zur Synchronisation verschiedener Prozesse einsetzen kann.

5.1.2 Einfache Schreib/Lese-Prozesse

Eine häufig benötigte Konstellation ist das Einlesen von Daten, die verarbeitet werden müssen und das anschließende Weiterleiten der Ergebnisse an eine andere Task. Dieses Verfahren ist typisch für die Pipeline-Verarbeitung. Gleichzeitig soll eine davon unabhängige Routine durchgeführt werden.

Das Programm *SimpleReadWrite* stellt eine einfache Lösung zu diesem Problem dar. Es besteht im wesentlichen aus vier Prozessen:

- ReadDataIn
- WorkOnData

- WriteDataOut
- WorkingProcess

Der ReadDataIn-Prozeß liest jeweils ein Datum über einen Stream von einer anderen Task. Die Prozedur WorkOnData soll gewisse Berechnungen mit dem durch ReadDataIn eingelesenen Datum durchführen und die Ergebnisse an den WriteDataOut-Prozeß weiterleiten, der diese an eine andere Task schickt. Im Working-Prozeß werden von den Daten unabhängige Operationen durchgeführt.

Zur Synchronisation der Prozesse werden die folgenden Semaphoren benötigt:

- ProtectDataIn, ProtectDataOut
- NewDataIn, NewDataOut
- PrgExit

```
MODULE SimpleReadWrite;
VAR   ProtectDataIn, ProtectDataOut : Semaphore;
      NewDataIn, NewDataOut, PrgExit : Semaphore;
      DataIn, DataOut : CARDINAL;

PROCEDURE ReadDataIn;
BEGIN
  LOOP
    Wait( ProtectDataIn );
    Lese 4 Bytes über einen Stream in die Variable DataIn;
    Signal( NewDataIn );
  END
END  ReadDataIn;

PROCEDURE WriteDataOut;
BEGIN
  LOOP
    Wait( NewDataOut );
    Schreibe die Variable DataOut über einen Stream an die Zieltask;
    Signal( ProtectDataOut );
  END
END  WriteDataOut;

PROCEDURE WorkOnData;
```

```
VAR  LocalData : CARDINAL;
BEGIN
  LOOP
    Wait( NewDataIn );
    LocalData ← DataIn;
    Signal( ProtectDataIn );
    ...              (* Es werden Berechnungen mit LocalData durchgeführt *)
    Wait( ProtectDataOut );
    DataOut ← LocalData;
    Signal( NewDataOut );
  END
END WorkOnData;

PROCEDURE WorkingProcess;
BEGIN
  ...
END WorkingProcess;

BEGIN
  InitSemaphore( ProtectDataIn, 1);
  InitSemaphore( ProtectDataOut, 1);
  InitSemaphore( NewDataIn, 0);
  InitSemaphore( NewDataOut, 0);
  InitSemaphore( PrgExit, 0);
  StartProcess( ReadDataIn );
  StartProcess( WriteDataOut );
  StartProcess( WorkOnData );
  StartProcess( WorkingProcess );
  Wait( PrgExit );
END SimpleReadWrite.
```

Die Abbildung 8 zeigt den Programmablauf innerhalb der Prozesse und deren Synchronisation über Semaphoren.

Die Semaphore `ProtectDataIn` dient zur Sicherung des eingelesenen Datums. Beim Verzicht auf `ProtectDataIn` könnten Daten verloren gehen, falls die Rate mit der die korrespondierende Task die Daten schickt sehr groß ist. Denn vor der Sicherung durch *LocalData* ← *DataIn* könnte der Wert von `DataIn` bereits durch einen neu eingelesenen Wert überschrieben worden sein. Daher muß der ReadDataIn-Prozeß solange mit dem Einlesen eines neuen Datums warten, bis der verarbeitende Prozeß den zuletzt gelesenen Wert nach `LocalData` gesichert hat. Ist dies der Fall, so setzt WorkOnData das entsprechende Signal über die Semaphore

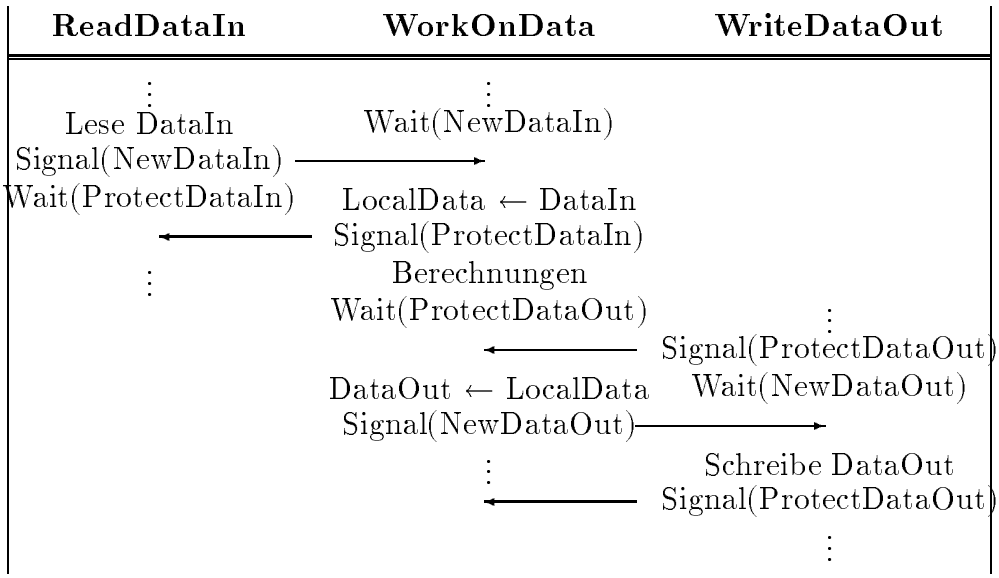

Abbildung 8: Die Synchronisation der Prozesse

`ProtectDataIn`.
Wird ein Datum von ReadDataIn gelesen, so setzt dieser Prozeß mit `NewDataIn` ein Signal, auf das der WorkOnData-Prozeß wartet.
Analog dazu wartet der NewDataOut-Prozeß auf ein Signal von WorkOnData, welches signalisiert, daß ein neues Ergebnis in Form von `DataOut` vorhanden ist. Zur Sicherheit setzt WriteDataOut nach erfolgreichem Verschicken das Signal `ProtectDataOut`. Analog zu `ProtectDataIn` schützt die Semaphore `ProtectDataOut` die Variable `DataOut` vor dem Überschreiben durch WorkOnData, falls das Verschicken des Ergebnisses mehr Zeit benötigt, als das Berechnen des nächsten Wertes.
Das Hauptprogramm wird erst beendet, wenn durch einen der Prozesse das `PrgExit`-Signal gesetzt wurde.

Die so programmierten Schreib- und Leseprozesse besitzen zwei wichtige Vorteile:

1. Während der Prozeß WorkOnData noch Berechnungen an dem alten Datum durchführt, kann durch ReadDataIn bereits ein neues Datum eingelesen werden. Analoges gilt für den Schreibprozeß. Weil der Datentransfer durch die Links realisiert wird, beansprucht dieser Vorgang fast keine Rechenzeit und man erhält eine Zeitersparnis zur Programmierung ohne Prozesse.[11]

[11] Lediglich das Betriebssystem Helios verursacht durch die benötigte *startup-Zeit* bei der Kommunikation eine gewisse Verzögerung.

2. Werden die Daten mit einer kleineren Rate gelesen als WorkOnData sie verarbeiten könnte, so haben der WriteDataOut sowie der WorkOnData-Prozeß häufig nichts zu tun und verbringen einen großen Teil der Zeit bei den Wait-Befehlen. Weil dieses Warten, wie bereits gezeigt, keinerlei Rechenzeit benötigt, steht dem Working-Prozeß die volle Rechenleistung zur Verfügung, da er in diesem Moment der einzige Prozeß des Prozeßverwalters ist.

Was passiert nun, wenn die Rate mit der die Daten gelesen werden können, größer ist als die Rate in der WorkOnData sie verarbeiten kann?
Weil der ReadDataIn-Prozeß nach dem Einlesen eines neuen Datums auf das Setzen der `ProtectDataIn`-Semaphore warten muß, verweilt der Lese-Prozeß längere Zeit beim Wait-Befehl. Dies ist nicht weiter schlimm, da der Working-Prozeß im Gegenzug mehr Rechenleistung zur Verfügung hat.

Falls aber die Rate mit der die Daten durch ReadDataIn gelesen werden nicht konstant, sondern gewissen Schwankungen unterworfen ist und die Verarbeitungszeit durch WorkOnData von Datum zu Datum verschieden ist, so kann man entstehende Wartezeiten durch die Implementierung eines Datenpuffers verhindern.

5.1.3 Schreib/Lese-Prozesse mit Datenpuffer

Der benötigte Datenpuffer kann als Array oder Liste angelegt werden. Jedes eingelesene Datum nimmt den Platz eines Elementes in dem Array bzw. in der Liste ein. Im Programm *BufferedReadWrite* wird ein Array als Puffer und die Variablen *FirstIn, LastIn* bzw. *FirstOut, LastOut* als Indizes für das erste und letzte noch nicht verarbeitete bzw. noch nicht geschriebene Datum benutzt.

Die Konstante *MaxBuffer* gibt die Länge des Puffers an. Anders als bei den einfachen Schreib/Lese-Prozessen werden hier die Semaphoren `ProtectDataIn` und `ProtectDataOut` mit der Puffergröße initialisiert und dienen zur Anzeige der Auslastung der Puffer. Die Semaphore `ProtectDataIn` gibt an, wieviel freie Plätze im Lesepuffer vorhanden sind, während `ProtectDataOut` die freien Plätze des Schreibpuffers zählt. Sollte der Wert einer dieser Semaphoren bei Null angelangt sein, so müssen der Lese- bzw. der Verarbeitungs-Prozeß solange beim Wait-Befehl warten, bis ein Pufferplatz frei geworden ist und die entsprechende Semaphore einen positiven Wert annimmt.

```
MODULE BufferedReadWrite;
CONST  MaxBuffer = 20;
VAR    ProtectDataIn, ProtectDataOut : Semaphore;
       NewDataIn, NewDataOut, PrgExit : Semaphore;
       BufferIn, BufferOut : ARRAY [1..MaxBuffer] OF CARDINAL;
```

```
        FirstIn, LastIn, FirstOut, LastOut : CARDINAL;

PROCEDURE ReadDataIn;
BEGIN
  LOOP
    Wait( ProtectDataIn );
    LastIn ← 1 + ( LastIn MOD MaxBuffer );
    Lese 4 Bytes über einen Stream in die Variable: BufferIn[LastIn];
    Signal( NewDataIn );
  END
END  ReadDataIn;

PROCEDURE WriteDataOut;
BEGIN
  LOOP
    Wait( NewDataOut );
    FirstOut ← 1 + ( FirstOut MOD MaxBuffer )
    Schreibe die Variable BufferOut[FirstOut] über einen Stream an die Zieltask;
    Signal( ProtectDataOut );
  END
END  WriteDataOut;

PROCEDURE WorkOnData;
VAR  LocalData : CARDINAL;
BEGIN
  LOOP
    Wait( NewDataIn );
    FirstIn ← 1 + ( FirstIn MOD MaxBuffer );
    LocalData ← Buffer[FirstIn];
    Signal( ProtectDataIn );
    ...              (* Es werden Berechnungen mit LocalData durchgeführt *)
    Wait( ProtectDataOut );
    LastOut ← 1 + ( LastOut MOD MaxBuffer );
    BufferOut[LastOut] ← LocalData;
    Signal( NewDataOut );
  END
END WorkOnData;

PROCEDURE WorkingProcess;
BEGIN
  ...
```

```
END WorkingProcess;

BEGIN
   FirstIn, LastIn, FirstOut, LastOut ← 0;
   InitSemaphore( ProtectDataIn, MaxBuffer);
   InitSemaphore( ProtectDataOut, MaxBuffer);
   InitSemaphore( NewDataIn, 0);
   InitSemaphore( NewDataOut, 0);
   InitSemaphore( PrgExit, 0);
   StartProcess( ReadDataIn );
   StartProcess( WriteDataOut );
   StartProcess( WorkOnData );
   StartProcess( WorkingProcess );
   Wait( PrgExit );
END BufferedReadWrite.
```

Wird die Größe des Puffers *MaxBuffer* auf den Wert 1 gesetzt, entspricht das Programm *BufferedReadWrite* dem ungepuffertem Lesen und Schreiben durch *SimpleReadWrite*.

Für viele anregende Diskussionen danke ich A.Bachem und W.Hochstättler. Das Projekt wurde gefördert durch das BMFT unter Projekt-Nr.17.

Literatur

[Ebe87] Ebert, H. : *Die Transputerbausteine.* c't (10)1987, 180–196.

[Far79] Farley, A.M. : *Minimal Broadcast Networks.* Networks 9(1979), 313–332.

[Gib88] Gibbons, A. / Rytter, W. : *Efficient Parallel Algorithms.* Cambridge 1988.

[Har90] Hardwig, T. : *Dinner for five.* c't 12(1990), 244–252.

[Inm1] INMOS Limited : *Transputer Reference Manual.* Prentice Hall 1988.

[Inm2] INMOS Limited : *Transputer Instruction Set, A compiler writer's guide.* Prentice Hall 1988.

[Per89] Perihelion Software : *The Helios Operating System.* Prentice Hall 1989.

[Spek88] Spektrum der Wissenschaft : *Die nächste Computer-Revolution.* Sonderheft 6, 1988.